L'art de programmer son copeau de silicium

4-AE/I release 2010a

1er octobre 2010

Table des matières

Chap 1: Architecture en couches

1.1 Introduction et définitions

Un pilote informatique (ou plus connu sous l'anglicisme « driver ») est un programme informatique qui permet à un autre programme d'interagir avec un périphérique. En général, chaque périphérique a son propre pilote.

Pour assurer la qualité d'une application, il est nécessaire de bien concevoir les pilotes et leur relation avec l'application. Petit rappel : en informatique, la qualité désigne un ensemble d'indicateurs pour offrir une appréciation globale d'un logiciel. Elle se base sur : la complétude des fonctionnalités, la précision des résultats, la fiabilité, la tolérance aux pannes, la facilité et la flexibilité de son utilisation, la simplicité, l'extensibilité, etc.

Parmi ces critères, le néologisme portabilité désigne, pour un programme informatique, sa capacité à fonctionner dans différents environnements d'exécution, en particulier différents environnements matériels. A cela s'ajoute l'adaptabilité qui est souvent utilisé pour désigner la qualité d'un logiciel qui peut être modifié aisément en harmonie avec les changements auxquels son utilisation peut être soumise.

La portabilité et l'adaptabilité d'une application nécessite d'avoir une architecture logicielle solide. Celle-ci définit l'organisation interne d'un logiciel, son découpage en couches et en modules, ainsi que les responsabilités de chaque module et la nature et la structure des relations entre modules.

Les relations entres les modules sont fournies via une interface de programmation (*Application Programming Interface ou API*). Elle permet l'interaction des programmes les uns avec les autres. Du point de vue technique une API est un ensemble de fonctions, procédures ou classes mises à disposition par une bibliothèque logicielle, un système d'exploitation ou un service. La connaissance des API est indispensable à l'interopérabilité entre les composants logiciels.

Dans le cas typique d'une bibliothèque, il s'agit généralement de fonctions considérées comme utiles pour d'autres composants. Une interface en tant que telle est quelque chose d'abstrait ; les composants réalisant celle-ci étant des mises en œuvre (ou implémentation). Idéalement il peut y avoir plusieurs mises en œuvre pour une même interface. Par exemple, sous UNIX, la libc définit des fonctions de base utilisées par pratiquement tous les programmes et est fournie par des mises en œuvre propriétaires ou libres, sous différents systèmes d'exploitation.

1.2 Architecture logicielle recommandée

Pour en revenir à nos moutons électroniques, nous préconisons pour les TP de périphériques une architecture logicielle en trois couches (voir figure 1.1).

Les différentes couches sont :

- **La couche application** : Cette couche ne comporte que du code dédié à une application. Elle comporte l'ensemble des fonctions de traitement et d'orchestration de l'application.
- **La couche de services applicatifs** : Cette couche un ensemble de fonctions qui masquent les périphériques matériels au niveau de l'application pour en offrir une représentation abstraite.

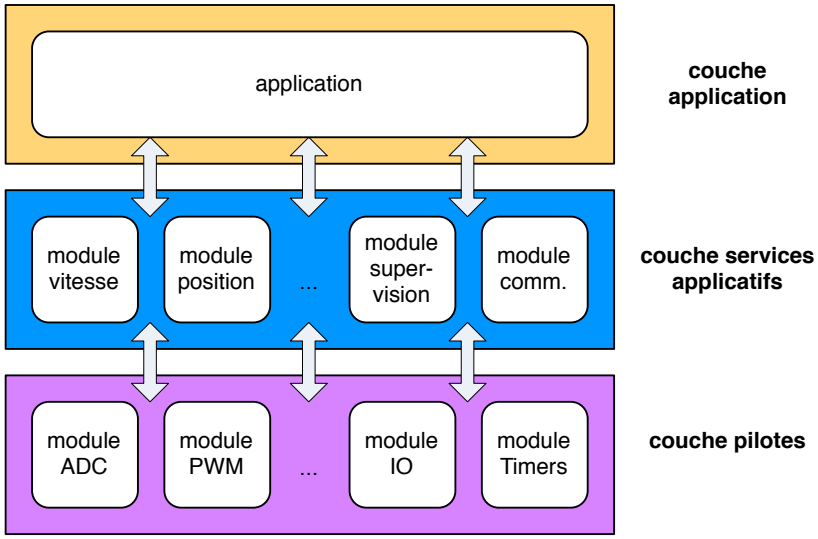


FIGURE 1.1 – Architecture logicielle

Par exemple, le service d'acquisition d'une vitesse masque l'appel au périphérique ADC pour offrir une vision purement abstraite de la variable de vitesse.

- **La couche de pilotes** : Cette couche ne comporte que du code lié à la configuration et l'utilisation des périphériques sans présupposé une utilisation particulière par une application.

Une telle architecture offre de nombreux avantages :

- Seules les couches « services applicatifs »et « pilotes »doivent changer si le matériel évolue, la couche application n'ayant a priori besoin que de quelques modifications pour prendre en considération les nouvelles capacités du support d'exécution (surtout en ce qui concerne le temps).
- Les modules qui composent la couche pilotes peuvent être utilisées pour différentes applications et ainsi offrir des composants pris sur étagères.
- La couche services applicatifs fournit des services génériques pour un même ensemble de familles d'applications sans avoir besoin d'être réécrit à chaque développement ou évolution d'une application sur le même support.
- Le test des modules est fait de manière unitaire, ce qui facilite le débogage sans avoir besoin de chercher la petite bête.

Les éléments qui composent une couche ne peuvent communiquer qu'avec un élément d'une couche inférieure et cela se fait uniquement à travers les API des différentes modules.

1.3 Guide des bonnes manières

Chaque module est associé à deux fichiers, l'un comportant le code (.c) et l'autre définissant son API (.h). Seul les fonctions déclarées dans le ficher d'en-tête sont accessibles aux autres éléments logiciels.

Comme le fichier d'en-tête est le seul élément utilisable (lisible / modifiable) par un utilisateur externe au module, il est capital de bien le commenter. Le commentaire est considéré comme un

mode d'emploi du module. Ceci est d'autant plus vrai, que les constructeurs de micro-contrôleurs proposent des modules (ou bibliothèques) au format .lib ou .o. Ces derniers sont compilés donc de fait, non modifiables.

Afin d'assurer la portabilité des modules de la couche pilotes, il est interdit qu'une des fonctions d'un module utilisent une variable globale au projet. Si on veut utiliser une variable globale au projet, dans la couches service ou application, cela peut se faire par le biais d'un fichier d'entête partagé, par exemple, global.h. Ceci étant, c'est fortement déconseillé.

Quelques règles sont spécifiques aux couches :

- **Couche application**
  - Le `main()` est contenu dans la couche application et ne peut faire appel qu'à des fonctions de la couche services.
  - Aucune référence à des périphériques n'est fait au niveau de la couche application.
- **Couche services applicatifs**
  - Le code des modules de la couches services contient des fonctions directements liées à l'application : les noms de fonctions et de variables font explicitement référence à l'application.
  - Les fonctions au niveau services appellent uniquement des fonctions de niveau pilotes. Elles ne s'appellent pas entre elles. Dit autrement, elles ne peuvent inclure que des fichiers de niveau pilote.
  - Les fichiers sont organisés par domaine lié à l'application
- **Couche pilotes**
  - La couche pilotes ne contient que les fonctions qui concernent directement les périphériques.
  - Les noms des fonctions doivent évoquer les périphériques.
  - Les fonctions ne font pas référence à l'application et doivent être définie de manière indépendante à toute application.

1.4 La couche matérielle

Le logiciel, et dans notre cas la couche pilote, est exécuté sur le coeur du processeur et doit communiquer avec les périphériques. Les périphériques sont des circuits, essentiellement de l'électronique numérique, présents sur la puce qui fournissent des fonctions d'interface avec le matériel sur lequel la puce est embarquée. Ces circuits ne sont pas programmables comme le coeur : ils ne peuvent pas exécuter des lignes de programmes. Par contre, on peut les configurer, c-à-d. modifier un paramètre de son fonctionnement, où transférer des données avec le coeur du processeur.

TODO SCHEMA REGISTRE GPIOA_ODR

Si l'on écrit une valeur à l'adresse XXX, avec la ligne `*(0x8000123)=13` en langage C qui sera traduite par exemple par `ldr R0,=0x8000123 ; ldr R1,=13; str R1,[R0]` en assembleur, le coeur va positionner la valeur $0x8000123$ sur le bus d'adresse, la valeur 13 sur le bus de donnée, la ligne $R/\overline{W}$ à 0 pour *write* et attendre un front d'horloge actif.

TODO finir tout ça

Chap 2: Faudrait pas prendre les pilotes pour des navigateurs !

Lors de l'implémentation d'un pilote plusieurs possibilités sont offertes en particulier pour tout ce qui est lié à la configuration.

2.1 Premier exemple : configuration d'une broche d'entrée-sortie

Prenons l'exemple de la configuration d'un port d'entrée-sortie. Au cour de la conception la fonction `Pin_IO_Init(State, Port, Pin_Number)` est décrite comme permettant au pin numéro `Pin_Number` du port `Port` d'être configurer dans l'état `State`.

Au niveau implémentation cela peut se traduire de deux manières : soit dynamiquement, c'est-à-dire que c'est à l'exécution que la configuration sera faite, soit statiquement, c'est-à-dire que la configuration sera faite à la compilation. Ces deux stratégies nécessitent alors une implémentation différente.

2.1.1 Pilote dynamique

Dans le cas dynamique, l'API du pilote fournit à l'application une fonction de configuration, par exemple l'API `./pilotes/GPIO_dyn.h` du listing 2.1 et son implémentation dans `./pilotes/GPIO_dyn.c`

```
// Port configuration function
void Pin_IO_Init(char State, GPIO_TypeDef * Port, int Pin_Number);
// State should be 'i' for INPUT, 'o' for OUTPUT
// Port pointer to GPIO structure defined in stm_regs.h : GPIOA, ...
// Pin_Number : as it says...
// Example :
// Pin_IO_Init('o',GPIOD,3); set port D pin number 3 as an output
```

Listing 2.1 – ./pilotes/GPIO_dyn.h : l'API du pilote dynamique

(listing 2.2) permet de régler les registre du `Port` voulus en fonction de `State` et `Pin_Number`.

La configuration d'une broche se fera par un appel de l'application (couche application ou service) en passant les paramètres souhaités à la fonction `Pin_IO_Init`. Dans l'exemple du listing 2.3, on régle la broche 2 du port GPIOA en entrée.

Les fonctions de configurations sont donc écrites dans le fichier .c, par exemple `./pilotes/GPIO_dyn.c` et on donne accès aux couches supérieures (service et application) via une API dument commentée, par exemple `./pilotes/GPIO_dyn.h`. L'arborescence des fichiers conseillée, voir fig. 2.1, permet ainsi de réutiliser le pilote dans plusieurs projets sans faire de copie multiple de ces fichiers.

```
void Pin_IO_Init(char State, GPIO_TypeDef * Port, int Pin_Number)
{
        if (State=='i')
        {// input port
                //reset mode[0:1] to 00 = floating input
                Port->CRL &= ~(3<<(Pin_Number*4));
        }
        else
        {//output port
                 //set mode[0:1] to 11 = output 50MHz
                Port->CRL |= (3<<(Pin_Number*4));
        }
}
```

Listing 2.2 – ./pilotes/GPIO_dyn.c : source du pilote dynamique

```
#include "../../../pilotes/GPIO_dyn.h"
// ou #include "GPIO_dyn.h" si le chemin
//       est déclaré dans les options du compilateur...
...
void main (void)
{
        Init_Clock();
        Init_GPIO();
        Init_Pin('i',GPIOA,2); // Bouton Valid de MCB167
}
```

Listing 2.3 – ./projet/tes_GPIO_dyn/Test_GPIO.c : source applicative

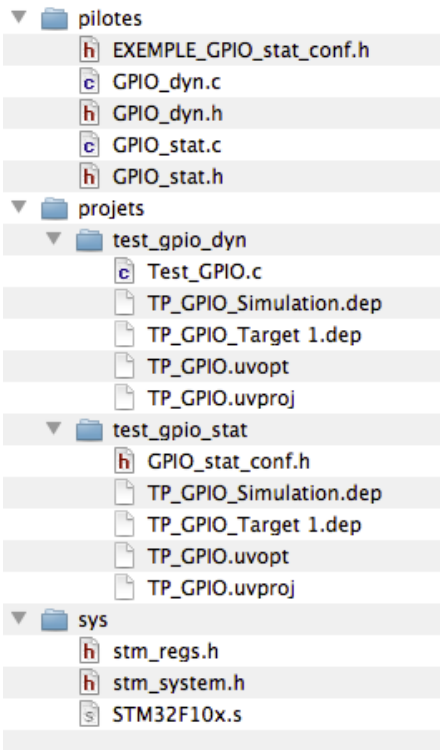


FIGURE 2.1 – Arborescence où le répertoire `pilotes` est au même niveau que celui des projets. Plusieurs projets peuvent donc inclure le même pilote en donnant le chemin relatif vers le .h (`../../pilotes/GPIO_stat.h`). Chaque applicatif de projet (`Test_GPIO.c`) configure son pilote en passant les paramètres lors d el'appel de la fonction d'initialisation. Dans le cas de pilote statique, le fichier de configuration `GPIO_stat_conf.h` est placé dans le projet car la configuration dépend de l'application et ne peut être la même.

2.1.2 Pilote statique

Dans le cas statique, la configuration se fait directement dans un fichier de configuration, par exemple `./projet/exemple/GPIO_stat_conf.h`, appartenant au projet. Dans ce fichier de configuration, l'utilisateur va commenter ou décommenter des directives `#define ...` :

```
//CONF : if you want to use GPIOA port then
// uncomment the following line
#define GPIOA_IS_USED

// CONF : by default all ports are inputs
// to set a port to output add lines in the format
// #undef P<port>_<pin>
// #define P<port>_<pin>  IS_OUTPUT
// Example : PB_3 fort pin number 3 of GPIOB
#undef PA_2
#define PA_2 IS_OUTPUT
```

Listing 2.4 – ./projet/exemple/GPIO_stat_conf.h : configuration via des directive de compilation

Le fichier de configuration est inclus par le fichier source, par exemple `./pilotes/GPIO_stat.c`, ce qui va influencer sa compilation et ainsi intégrer les configurations :

L'application, quand à elle, ne fait qu'appeler une fonction d'initialisation sans paramètres :

Dans cet exemple la simplicité n'est pas en faveur de la version statique, cependant le code

```
#define IS_INPUT 0x0
#define IS_OUTPUT 0x3

#define PA_0 IS_INPUT
#define PA_1 IS_INPUT
#define PA_2 IS_INPUT
...

#include "GPIO_stat_conf.h"
// path to the conf file should be set in compiler options
// Now PA_x are configured by user

void GPIO_Init(void)
{
   #ifdef GPIOA_IS_USED
   // construct the init from the PA_x configured
   GPIOA->CRL = (PA_0<<0)|(PA_1<<4)|(PA_2<<8)|(PA_3<<12)|(PA_4<<16);
   #endif
}
```

Listing 2.5 – ./pilotes/GPIO_stat.c : configuré via des directives de compilation

```
#include "../../../pilotes/GPIO_stat.h"
// ou #include "GPIO_stat.h" si le chemin est dans les options...
...
void main (void)
{
        Init_Clock();
        Init_GPIO();
}
```

Listing 2.6 – ./projet/exemple/main_stat.c : source applicative

produit statiquement est beaucoup moins coûteux en terme de mémoire, car il ne compilera les fonctions de configuration que si des ports sont utilisés.

2.2 Second exemple : configuration d'une interruption

Les interruptions peuvent permettre d'exécuter une fonction liée à l'application suite à un événement matériel. Il y a donc d'un côté la fonction qui appartient à la couche applicative et de l'autre l'interruption à capturer au niveau de la couche pilotes. Lors de la configuration du système il faut être capable de faire le lien entre ces deux éléments...

2.2.1 Pilote dynamique

Dans sa version dynamique, cela revient à prototyper une fonction dans le pilote du périphérique qui permet de faire le lien entre la fonction et une interruption du périphérique. Supposons que nous voulons exécuter la fonction `toto(void)` lors de l'interruption provoquée par le périphérique `XXX`.

Le pilote de `XXX` comportera une fonction `Init_IT_XXX(void (* IT_fonction) (void))` qui permettra de lier l'interruption de `XXX` à la fonction `IT_fonction`.

```
void (*pt_IT_Hook) (void); // Pointeur de fonction pour l'IT

void Init_IT_XXX(void (* IT_fonction) (void))
{
        pt_IT_Hook = IT_fonction;
        ... // Activer l'interruption
}
...
```

Listing 2.7 – ./pilotes/Pilote_XXX.c : configuration d'une interruption

Le handler de l'interruption est alors défini dans la suite de manière générique par la code suivant :

```
...
void XXX_IRQHandler void // Handler de l'IT de XXX
{
        if ((int) pt_IT_Hook != 0)
        {
                (*pt_IT_Hook)(); // Appel à la fonction
        }
}
```

Listing 2.8 – ./pilotes/Pilote_XXX.c : handler d'une interruption en dynamique

Au niveau applicatif la configuration du périphérique se fera grâce au code suivant :

Bien sûr la fonction de configuration de l'interruption pourrait être plus évoluée, par exemple en fixant le niveau de l'interruption etc.

```
#include "../../../pilotes/Pilotes_XXX.h"
...
void main (void) // Handler de l'IT de XXX
{
        Init_Clock();
        Init_GPIO();
        Init_IT_XXX(&toto);
}

void toto(void)
{
        // Code à exécuter pendant l'interruption
}
...
```

Listing 2.9 – ./projet/ex_it/main.c : Exemple de configuration d'une interruption en dynamique

2.2.2 Pilote statique

Dans le cas d'un pilote statique la solution est plus simple à implémenter. Il suffit de faire un fichier de configuration :

```
// CONFIGURATION PART
// please uncomment if you wich to launch a function on IT
// #define THERE_IS_A_HOOK_ON_IT
#ifdef THERE_IS_A_HOOK_ON_IT
        // CONFIGURE here the function prototype
        void toto (void);
        //CONFIGURE here the call to the hook function
        #define IT_HOOK_CALL  toto()
#endif
```

Listing 2.10 – ./projet/exemple/Pilote_XXX_conf.h : configuration via des directive de compilation

Le source du pilote utilise la directive `#ifdef` pour insérer ou non l'appel de la fonction lors de l'interruption

Remarquez que dans cet exemple, la version statique va générer uniquement un *Handler* vide alors que la version dynamique génère un *Handler* avec 4 lignes de code, plus une fonction d'initialisation (1 ligne) et son appel du main (1 ligne). Vous comprenez maintenant la compacité de la version statique mais malheureusement aussi sa complexité...

À vous de choisir...

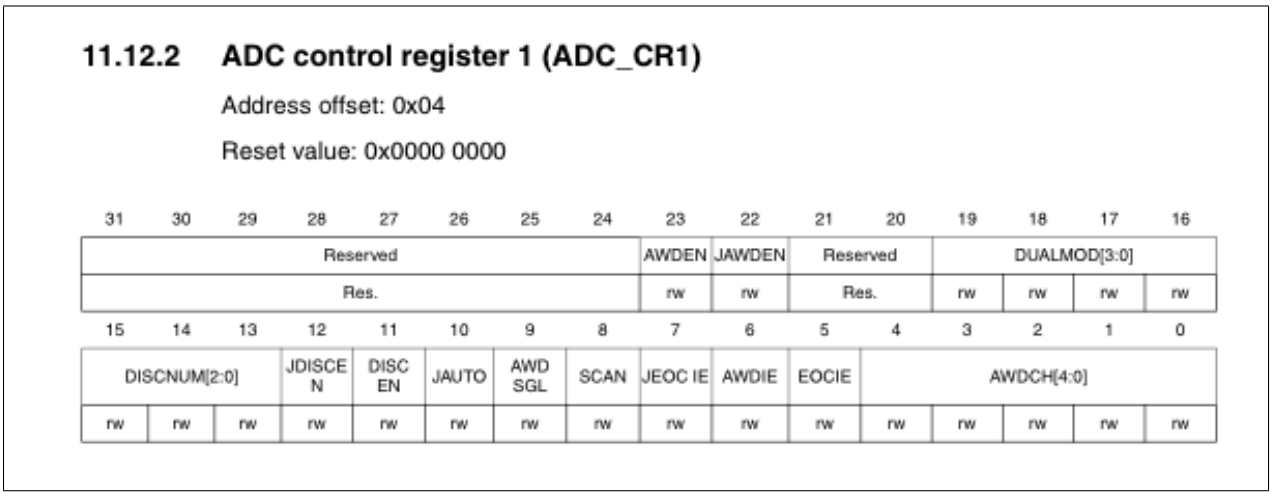
```
#include "Pilote_XXX_conf.h"
// path to the conf file should be set in compiler options

void XXX_IRQHandler void // Handler de l'IT de XXX
{
        #ifdef THERE_IS_A_HOOK_ON_IT
                IT_HOOK_CALL;
        #else
                ...
        #end
        ...
}
```

Listing 2.11 – ./pilotes/Pilote_XXX.c : handler d'IT utilisant les directives de compilation

Chap 3: Bah ! Les masques

Pour programmer un périphérique il est nécessaire d'aller modifier un ou plusieurs bits dans un registre sans modifier les autres. Par exemple le registre `ADC_CR1` sert à configurer le convertisseur analogique–numérique `ADC1` :



11.12.2 ADC control register 1 (ADC_CR1)

Address offset: 0x04

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved								AWDEN	JAWDEN	Reserved		DUALMOD[3:0]			
Res.								rw	rw	Res.		rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DISCNUM[2:0]			JDISCEN	DISCEN	JAUTO	AWDSGL	SCAN	JEOCIE	AWDIE	EOCIE	AWDCH[4:0]				
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

FIGURE 3.1 – Extrait du *reference manual* du STM32 p. 236, la suite décrit la fonction de chaque bit, comme `SCAN` et `AWDIE` et la signification des valeurs de chaque tranches de bit tels que `DISCNUM`

Pour cela on utilise les masques logiques et certaines astuces pour construire un masque clairement sans risquer de se tromper. Dans l'exemple suivant le bit `SCAN` de `ADC_CR1` est mis à 1 et le bit `EOCIE` est mis à 0 sans toucher aux autres bits. Les registres `ADC1_CR2`, `ADC1_SQR1`, `ADC1_SQR3` sont aussi manipulés avec des masques :

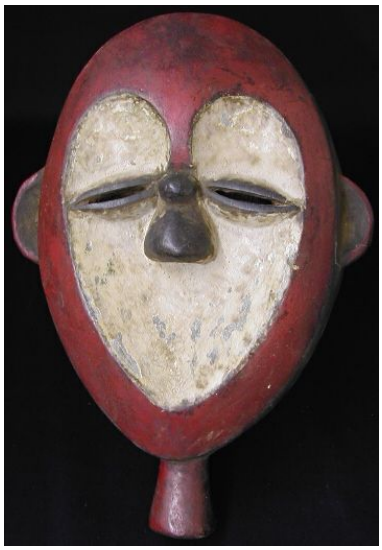
```
ADC1->CR1  |= (ADC_SCAN);      // continuous scan of channels 1,14,15

ADC1->CR1  &= ~(ADC_EOCIE);    // pas d'interruption de fin de conv.

ADC1->CR2  |= (ADC_EXTSEL_ON_SWSTART | ADC_ CONT | ADC_DMA);
    // EXTSEL = SWSTART
    // use data align right, continuous conversion
    // send DMA request

// convert sequence is channel 1 then 14 then 15
ADC1->SQR3  |= (1 <<SQ1_SHIFT)  |  (14 <<SQ2_SHIFT) | (15 <<SQ3_SHIFT);
```

Listing 3.1 – Extrait de la configuration de l'ADC de la baguette magique vue en assembleur



Si vous avez aucune idée de comment ce code fonctionne c'est que vous ne maitrisez pas encore l'art primitif du masque : lisez–donc la suite.
Ci–contre, un masque primitif (on préfère dire d'art premier*) d'origine Gabonaise.*

3.1 Opérateur logiques et bit à bit

On utilise les opérateurs logiques ET, OU, XOR (OU exclusif) et NOT entre un registre et un masque pour manipuler les bits. Les opérateurs logiques et leurs syntaxe en langage C sont résumés dans le tableau suivant :

Fonction logique	Opérateur logique	Opérateur bit à bit
ET	&&	&
OU	||	|
XOR	aucun	^
NON	!	~

L'opérateur logique considère les opérandes (qu'elle soient 8/16/32 bit) comme une valeur booléenne fausse si tous les bits sont nuls et vrai sinon. Elle fournit un résultat booléen nul si c'est faux et différent de zéro (en général la valeur 1) sinon. Ne confondez donc pas l'opérateur logique avec l'opérateur bit à bit qui effectue 8/16/32 opérations logiques entre chaque bits respectifs des opérandes, par exemple :

```
a = 2; // soit b00000010 en binaire est vrai car différent de 0
b = 1; // soit b00000001 en binaire est vrai aussi
y = a && b; // donne 1(vrai) car a ET b est vrai
z = a & b; // donne b00000000 car chaque ET des bits de a et b sont
    faux
```

3.2 Mettre un bit à 1

Pour cela on utilise l'opérateur OU avec une valeur binaire, appelée *masque*, ayant des bits à 1 uniquement devant les bits que l'on veut initialiser :

	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
OU	0	0	0	1	0	0	1	1
$=$	b_7	b_6	b_5	1	b_3	b_2	1	1

car $x\ OU\ 1 = 1$ et $x\ OU\ 0 = x$

Ainsi les bits b_4, b_1 et b_0 sont passés à 1 sans modifier la valeur des autres bits.

Pour effectuer cela en langage C on doit calculer la valeur du masque binaire : convertir `b00010011` en hexadécimal (`0x13`) ou en décimal (19) car le langage C n'admet pas de littéraux en binaire.

```
char avoile;
...
//formulations équivalentes
avoile = avoile | 0x13;    // opérateur | inline
avoile |= 0x13;            // opérateur | préfixé à =
avoile |= 19;              // valeur du masque en décimal
```

Il n'est pas très évident de comprendre que `0x13` ou `19` correspond à un masque visant les bits de rang 0,1 et 4, de plus il est très facile de se tromper lorsque l'on fait la conversion soi–même. Un *geek* utilisera l'opérateur de décalage à gauche `<<x` pour positionner un 1 devant le bit de rang x pour construire sont masque ainsi :

			b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
	$(1 << 0)$	$\rightarrow$	0	0	0	0	0	0	0	1
OU	$(1 << 1)$	$\rightarrow$	0	0	0	0	0	0	1	0
OU	$(1 << 4)$	$\rightarrow$	0	0	0	1	0	0	0	0
$=$	$(1 << 0)\|(1 << 1)\|(1 << 4)$	$\rightarrow$	0	0	0	1	0	0	1	1

Ainsi le code suivant est plus lisible et ne risque pas de comporter d'erreur de conversion :

```
//formulations équivalentes
avoile = avoile | 0x13;    // opérateur | inline
avoile = avoile | (1<<0)|(1<<1)|(1<<4);
avoile |= (1<<0)|(1<<1)|(1<<4);
```

3.3 Mettre un bit à 0

Pour cela on utilise l'opérateur ET avec une masque ayant des bits à 0 uniquement devant les bits que l'on veut initialiser :

$$\begin{array}{lcccccccc} & b_7 & b_6 & b_5 & b_4 & b_3 & b_2 & b_1 & b_0 \\ ET & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ \hline = & b_7 & b_6 & b_5 & 0 & b_3 & b_2 & 0 & 0 \end{array} \quad \text{car} \quad x\ ET\ 1 = x \quad \text{et} \quad x\ ET\ 0 = 0$$

Ainsi les bits b_4, b_1 et b_0 sont passés à 0 sans modifier la valeur des autres bits.

Pour construire le masque, on peut toujours effectuer la conversion soit-même avec un risque d'erreur : $b11101100 = 0xEC = 236$. On peut aussi construire le masque avec des 1 devant les bits à annuler et ensuite inverser chaque bit avec l'opérateur `~` .

$$\begin{array}{lcccccccccc} & & (1<<0)|(1<<1)|(1<<4) & \rightarrow & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ & \sim & (1<<0)|(1<<1)|(1<<4) & \rightarrow & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ ET & & & \text{avoile} & \rightarrow & b_7 & b_6 & b_5 & b_4 & b_3 & b_2 & b_1 & b_0 \\ \hline = & \text{avoile} \ \& \ \sim & ((1<<0)|(1<<1)|(1<<4)) & \rightarrow & b_7 & b_6 & b_5 & 0 & b_3 & b_2 & 0 & 0 \end{array}$$

Ce qui donne en langage C :

```
//formulations équivalentes
avoile = avoile & 0xEB;
avoile = avoile & ~((1<<0)|(1<<1)|(1<<4));
avoile &= ~((1<<0)|(1<<1)|(1<<4));
```

3.4 Inverser un bit

Pour cela on utilise l'opérateur XOR avec une masque ayant des bits à 1 uniquement devant les bits à inverser :

$$\begin{array}{lcccccccc} & b_7 & b_6 & b_5 & b_4 & b_3 & b_2 & b_1 & b_0 \\ XOR & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ \hline = & b_7 & b_6 & b_5 & \overline{b_4} & b_3 & b_2 & \overline{b_1} & \overline{b_0} \end{array} \quad \text{car} \quad x\ XOR\ 1 = \overline{x} \quad \text{et} \quad x\ XOR\ 0 = x$$

Ainsi seuls les bits b_4,b_1 et b_0 ont été inversés.

On construit les masques comme les masque OU de mise à 1.

3.5 Initialiser une tranche de bits

Une tranche de bit est un ensemble de quelques bits contigus dont la valeur a une signification particulière. Par exemple `DISCNUM` est une tranche de 3 bits du registre `ADC_CR1` qui indique le nombre de canaux à convertir.

Un programmeur avertis peut désirer initialiser une tranche de `avoile` à la valeur contenue dans `numero`. Pour cela il faut procéder en trois étapes : limiter la valeur de `numero` pour ne pas dépasser la tranche de 3 bits et la caler au bon endroit ; annuler la tranche de trois bits de `avoile` ; puis recopier les 1 du premier masque dans `avoile` avec un OU :

	num	$\rightarrow$	n_7	n_6	n_5	n_4	n_3	n_2	n_1	n_0
	$0x7$	$\rightarrow$	0	0	0	0	0	1	1	1
	$(\text{num} \,\&\, 0x7)$	$\rightarrow$	0	0	0	0	0	n_2	n_1	n_0
	$(\text{num} \,\&\, 0x7) << 4$	$\rightarrow$	0	n_2	n_1	n_0	0	0	0	0
OU	avoile $\&\ \sim(0x7 << 4)$	$\rightarrow$	b_7	0	0	0	b_3	b_2	b_1	b_0
=	$(\text{avoile} \,\&\, \sim(0x7 << 4)) \,\vert\, (\text{num} \,\&\, 0x7) << 4$	$\rightarrow$	b_7	n_2	n_1	n_0	b_3	b_2	b_1	b_0

Le programme suivant permet d'affecter la tranche `DISCNUM` avec la valeur contenue dans la variable `numero` :

```
void set_discnum ( char numero)
{
   // raz de la tranche DISCNUM : 3 bits de rang 13-15 de ADC_CR1
   ADC->CR1 &= ~(0x7<<13); // seuls les bits 13-15 du masque valent 0

   // init de la tranche avec numero
   ADC->CR1 |= numero<<13 // recopie les 1 de numero décalé au rang 13

   //Attention si numero >7 ça déborde sur les bits 16 à 31
   //Un masque doit limiter numero de 0 à 7 : pas plus de trois bits à
      1
   // On préfère donc cette ligne
   ADC->CR1 |= (numero & 0x7) <<13 // numero est limité
 }
```

Chap 4: L'épointeur de fonctions ne manque(nt) pas de piquant !

4.1 Appel indirect des fonctions

Extraits de : T. Monteil *et al.*, Du langage C au C++, Presses Universitaires du Mirail, 2009.

De la même manière que le nom d'un tableau représente l'adresse d'implantation de ce tableau (c'est-à-dire l'adresse de son premier élément), le nom d'une fonction (sans parenthèses) représente l'adresse de cette fonction, plus exactement l'adresse de son point d'entrée. Cette adresse peut être :

- attribué à un pointeur (pointeur de fonction), ce qui permet l'appel indirect de cette fonction,
- transmise comme paramètre à d'autres fonctions,
- placée dans un tableau de pointeurs de fonction.

4.1.1 Pointeur de fonction

Un pointeur de fonction est une variable pouvant recevoir l'adresse d'une fonction. Dans la déclaration du pointeur, cette fonction est typée, si bien qu'il ne peut ensuite accepter de recevoir que l'adresse d'une fonction de même type. Voici la forme générale de la déclaration d'un tel pointeur :

```
type ( * ptrFct ) (type, type, ...);
```

Les parenthèses de gauche sont obligatoires, sinon il s'agirait d'une fonction retournant un pointeur.

4.1.2 Affectation d'un pointeur de fonction

Voici comment on affecte l'adresse d'une fonction à un pointeur :

```
double ( * Math ) ( int, int, double ) ; /* Pointeur de fonction */
double Racine ( int, int, double ) ; /* Fonction */

Math = Racine ; /* Affectation */
```

4.1.3 Appel de fonction par un pointeur

Dès qu'un pointeur à reçu l'adresse d'une fonction, on peut appeler celle-ci indirectement selon la forme générale suivante :

```
/* définition d'un pointeur */
type ( * ptrFcn ) (type, type, ...);
```

```
/* déclaration d'une fonction de même type renvoyé */
type Fonc (type, type, ...);

/* affectation du pointeur */
ptdFcn = Fonc;

/* appel indirect de Fonc() */
(*ptrFcn) (arguments);
```

Dans le cas présent, les deux instructions ci-dessus sont bien équivalentes à un appel direct de `Fonc()` puisque `*ptrFcn` a pour valeur `Fonc`.

4.1.4 Passage de fonctions en paramètre

Il est possible de transmettre une fonction en paramètre d'une fonction appelante. Celle-ci emploie le nom de la fonction appelée sans parenthèses ni arguments à la suite. C'est donc l'adresse du point de lancement de cette fonction qui est empilée. Pareillement, on peut transmettre indirectement une fonction en paramètre grâce à un pointeur sur cette fonction. Dans l'exemple qui suit, `Fonction()` est appelée deux fois avec deux (adresses de) fonctions différentes passées en argument, et une fois avec un pointeur de fonction :

```
int Premiere (void) ;
int Seconde (void) ;
int ( * pFonction ) () = Seconde ;

void Fonction (int (* ptrFoncArg) (), int Un , int deux )
{
/* Corps de Fonction () */
}
/* exemple d'appels */
/*****************************************************************************/
/* Appel directement avec l'adresse de la fonction Premier et les entiers 12 et -45 */
Fonction ( Premiere, 12, -45);

/*****************************************************************************/
/* Appel avec l'adresse de la fonction Premier et les entiers 0 et 0               */
Fonction ( Seconde, 0, 0);

/*****************************************************************************/
/* Appel indirect avec l'adresse de la fonction Seconde contenue                   */
/* dans le pointeur de fonction */
/* Les deux entiers sont affecté avec la sortie des fonctions Premiere et Seconde   */
Fonction ( pFonction, Premiere (), Seconde () );
```

4.2 L'art et la manière...

Pour bien comprendre l'utilité de cette technique de programmation, il convient d'en rappeler l'objectif principal : **nous voulons offrir le moyen à un développeur d'application de configurer un périphérique sans qu'il est besoin d'en connaître son fonctionnement**. Le code que produit le développeur est uniquement dans la couche application (voir chapitre 1) et il ne peut faire appel qu'à des fonctions de la couche pilote pour manipuler le matériel.

Il se pose alors le problème suivant : comment offrir un service à un développeur d'applications qui lui permette d'exécuter lors d'une interruption une fonction qu'il aura développée ?
Il est à noter que cette approche est équivalente à la notion de services qu'offre un système d'exploitation.

Par exemple, il souhaite exécuter la fonction :

```
void Ma_Fonction_IT ( void )  /* Couche application */
{
   /* Le code à exécuter pendant l'interruption */
}
```

Il faut noter que, quoiqu'il en soit, cette fonction ne pourra ni prendre d'arguments en entrée ni en retourner puisque son appel ne sera pas déclencher logiciellement.

Il faut donc que dans le Handler de l'interruption (la fonction qui sera appelée en premier lieu lors de l'interruption) il fasse appel à cette fonction, soit :

```
void XXX_IRQHandler ( void )  /* Couche pilote */
{
  Ma_Fonction_IT ();
}
```

Or cela ne respecte pas nos règles de codage, puisque le Handler doit faire partie de la couche pilote alors que `Ma_foncion_IT` fait partie de la couche application...

Il n'y a pas de solution miracle, il faut faire un peu de programmation « avancée » ! Ca fait du bien aux neurones... Remarquons au passage que la solution est fournie dans le chapitre 2.

Si d'autres solutions — effectives — sont proposées, nous sommes preneur.

4.2.1 ... de s'en servir avec la solution dynamique

Une première solution consiste à utiliser un pointeur de fonctions et s'écrit :

```
/* Fichier de la couche "application" */
void Ma_Fonction_IT ( void )
{
  /* Le code à exécuter pendant l'interruption */
}

/* à faire dans le main */
Init_Periph(Ma_Fonction_IT);
```

```
/* Fichier de la couche pilote */
void (* pFnc) (void) ; /* déclaration d'un pointeur de fonction */

void XXX_IRQHandler ( void )
{
  if (pFnc != 0)
    (*pFnc) (); /* appel indirect de la fonction */
}

Init_periph (void (* ptrFonction) (void))
{
  pFnc = ptrFonction; /* affectation du pointeur */
}
```

4.2.2 ... de s'en passer avec la solution statique

Une seconde solution consiste à définir dans un fichier de configuration la fonction à exécuter pendant l'interruption :

```
/* Fichier de la couche application écrite par l'utilisateur du pilote*/
void Ma_Fonction_IT ( void )
{
  /* Le code à exécuter pendant l'interruption */
}

/* Fichier de configuration modifiable par l'utilisateur du pilote*/
#ifdef HOOK_ON_XXX
  #define XXX_HOOK_CALL Ma_Fonction_IT ()
#endif

/* Fichier de la couche pilote uniquement visible par le développeur du pilote */
#ifdef HOOK_ON_XXX
void XXX_IRQHandler ( void )
{
    XXX_HOOK_CALL ;
}
#end
```

4.2.3 Quelques commentaires pour se faire une religion

Ces deux solutions sont élégantes et permettent de bien séparer les couches application et pilote. Elles ne sont pas strictement identiques au niveau fonctionnel et comportent quelques nuances :

- La version dynamique permet de configurer l'interruption à la volée et de la reconfigurer en cours d'exécution.
- La version statique nécessite une compilation du fichier pilote et donc de fournir son code, ce qui n'est pas le cas de la version dynamique. De plus, il faut maintenir le fichier de configuration.

– La version statique va générer uniquement un Handler s'il y a une fonction à exécuter alors que la version dynamique génère un Handler avec 4 lignes de code, plus une fonction d'initialisation (1 ligne) et son appel du main (1 ligne), ce qui permet d'avoir un code plus compact.